A Software Architecture for Intelligent Synthesis Environments[1]

Robert E. Filman
Research Institute for Advanced Computer Science
NASA Ames Research Center MS/269-2
Moffett Field, CA 94035
650–604–1250
rfilman@mail.arc.nasa.gov

Abstract—The NASA's Intelligent Synthesis Environment (ISE) program is a grand attempt to develop a system to transform the way complex artifacts are engineered. This paper discusses a "middleware" architecture for enabling the development of ISE. Desirable elements of such an *Intelligent Synthesis Architecture* (ISA) include remote invocation; plug-and-play applications; scripting of applications; management of design artifacts, tools, and artifact and tool attributes; common system services; system management; and systematic enforcement of policies. A typical middleware foundation for an ISA is a distributed object technology such as CORBA (Common Object Request Broker Architecture). I argue that such an architecture can be profitably extended by enabling "plug-and-play" insertion of new policies into the system. I describe the Object Infrastructure Framework, an Aspect Oriented Programming (AOP) environment for developing distributed systems that provides policy insertion. This technology can be used to enforce policies such as maintaining the annotations of artifacts, particularly the provenance and access control rules of artifacts; performing automatic datatype transformations between representations; supplying alternative servers of the same service; reporting on the status of jobs and of the system; conveying privileges throughout an application; supporting long-lived transactions; maintaining version consistency; and providing software redundancy and mobility.

TABLE OF CONTENTS

1. INTELLIGENT SYNTHESIS ENVIRONMENT

The NASA's Intelligent Synthesis Environment (ISE) program is a grand attempt to develop a system to transform the way complex artifacts are engineered. The program "aims to link scientists, design teams, manufacturers, suppliers and consultants in the creation and operation of an aerospace system" and seeks "An end-to-end simulation of the product life cycle" [1]. That is, using ISE, geographically distributed engineers and scientists creating the design for, say, a spacecraft, should be able to easily collaborate. Such collaboration would include not only sharing information, but using advanced analysis systems and virtual reality environments to explore the properties of different designs. ISE should track the changes and versions of design artifacts and protect the intellectual property of the participants. The ISE vision thus shares intent of applying analysis tools to designs with other initiatives (e.g., DARPA's Simulation-Based Design [2] and the National Industrial Information Infrastructure Protocols Consortium's STEP standard [3]).

Structurally, the ISE program divides into groups concerned with Collaboration (tools for sharing information), Human-Centered Computing (immersive environments), Life Cycle Integration and Validation (version management) and Infrastructure for Distributed Computing (software architecture). ISE is a distributed application about manipulating and analyzing design artifacts. It is natural to think of the system as dividing into three elements

- The **underlying communication structure** (network, network transportation protocols), which reliably and quickly moves bits from one distributed location to another.

- The **application code**, which gathers the users' intent (e.g., taking commands, be they typed text or hand-squeezes in virtual space), performs actions asserted by that intent (e.g., executing analysis systems or computing the representation of a deformation under pressure), and displays the results back to the user (e.g., updating the display of a cathode ray-tube or providing pressure resistance in a pneumatic glove).

- The **middleware layer**, which simplifies the task of coding applications by providing uniform interfaces, and useful services, and by systematically preserving the policies of the overall application.

[1] 0-7803-6599-2/01/$10.00 © 2001 IEEE

2. Intelligent Synthesis Architecture

This paper is concerned with that middleware layer: what abstractions can we provide to ISE application builders to simplify their work. We call such an organization an Intelligent Synthesis Architecture (or ISA, for short). Desirable elements of an ISA include:

- **Remote invocation.** Applications should be able to transparently invoke other applications and services. Remote invocation should be as simple as possible—the more we can relieve the application programmer from the task of thinking of remote elements as being different than local elements, the better.

- **Plug-and-play tools.** We can't anticipate all the applications in the ISE. Comprehensive analysis will not resolve this problem: Some applications will be developed only after system deployment. We need to be able to dynamically add new applications to the system. More generally, we need "plug-and-play" tools, where a *tool* is any invocable element in the system. Examples of tools include *applications,* such as analysis programs on designs (e.g., a computational fluid dynamics program), *services* provided by the architecture invocable by applications (e.g., a messaging service), and *scripts*, composite tools glued together by descriptions of processes of tool invocations.

- **Scripting.** Invoking a single tool will not be enough to accomplish most high-level tasks. The ISA needs to provide facilities for expressing compositions of tool activities—effectively, scripting languages for chaining together tool invocations.

- **Management of design artifacts.** The goal of an ISE is to produce designs, and the ISA must provide a repository of design elements.

- **Management of design artifact attributes.** As important as it is to have actual designs is to know what it is we have. Therefore, a critical element of an ISA is a repository of annotations about design artifacts. In contrast to "fully designed systems" we don't believe we will know ahead of time all the varieties of annotations which applications will want to apply to design artifacts. Thus, we need a flexible repository of annotating design artifacts (and, for that matter, the other elements of the system, such as tools and users).

- **Management of tools and tool attributes.** With the repositories of tools, designs and annotations of tools and designs comes the concomitant need to manage the space of tools, designs and attributes and the need for systems to facilitate that management. Of particular importance in an ISA are keeping track of the versions of system elements, the configurations of composite system elements, and the pedigree of design elements

- **Services**. Services are common (or not so common) facilities provided by an ISA under the assumption that application programs will find them useful. Examples of such services include name services, matchmaking, and trading (white and yellow page mechanisms for finding references to objects with particular names or properties), event and publish-and-subscribe mechanisms for asynchronous invocation, tool version and configuration management and location, transactions; and agent infrastructure (facilities for moving executing systems closer to their data and to simplify the communication and coordination among executing components.)

- **System management**. A complex system also needs facilities for tracking system usage and for discovering, analyzing and recovering from faults.

- **Systematic enforcement of policies**. Federated and heterogeneous systems require interesting political arrangements. The owners of an ISE will want certain policies obeyed throughout the system—requirements about elements such as access control, authentication, priorities, intrusion detection, and reliability. The ISA should enforce these policies with minimal effort on the part of the application programmers.

Fortunately, many of these elements already exist. Distributed object technologies such as CORBA (Common Object Request Broker Architecture) and EJB (Enterprise Java Beans) provide transparent remote invocation, plug-and-play tools and some services and system management. Such systems also provide some system management facilities and a few specific policies that application programs can apply.

Similarly, scripting languages abound. There are clearly opportunities for describing scripting systems particularly amenable to ISE. In particular, ISE has the need for scripting systems that incorporate and enforce processes (that is, combining the ability to perform action sequentially with the need for workflow to coordinate the activities of distributed collaborators) and the ability to script systems for variational design (that is, to search for designs that are optimal with respect to specific criteria.) This issue is clearly fodder for a paper, but not this one.

Systems such as Product Data Management (PDM) systems currently store complex design information, particularly CAD/CAM/CAE design information. ISE's repository demands resemble PDMs in needing to store complex data, but lack the clear PDM knowledge of what kinds of things are being stored and what needs to be known about them. We believe that we need to store not only design artifacts and tools, but also meta-information about data and programs. For example, for a given data set, applications may need to know who created it, by executing which process, at what time, using which versions of which applications, how long it took to create, what design assumptions went into choosing to run the program, and so forth. Such information can be used, for example, to track the consequences of faulty data or programs or to recognize when different designers have conflicting concepts of what makes one system preferable to another. We note three things about this meta data: (1) the ISA layer is not going to know exactly which annotations will prove useful to which artifacts, (2) that it can't be cumbersome to introduce new annotations, and (3) the creator of a particular tool or script may not be the one who needs the creation of a particular annotation.

Enforcing consistent annotations of data and tools are thus the first illustration of a systematic policy that needs to be applied throughout the ISE. In the remainder of this paper, we discuss a computational mechanism for implementing such policy consistency.

3. Aspect-Oriented Programming

Developing a computer system involves keeping cognizant of and programming elements to deal with a variety of concerns. Beyond the base functionality of the system, most applications must deal with "ilities" such as security, reliability, manageability, quality of service, concurrency, and so forth. These clutter what might have been a straightforward program with calls to orthogonal services. Such elements are in the code because they perform necessary activities, but they work against the creation of correct and maintainable systems. The application programmer, wise in the world of thermal simulation or computational fluid dynamics is unlikely to be able to correctly program distributed synchronization. Even having been provided synchronization methods and taught when to call them, she is still likely to mistakenly omit them or use them incorrectly. And down the road, a programmer called upon to maintain that system would likely be mystified by these little intrusions. He would have no idea when to preserve, delete or insert them in modifying the system.

Ideally, one would like to separate out the specification and coding of each of these separate concerns and have some automatic mechanism "weave" the separate concerns back together into a working system. This is the premise underlying the work in the emerging disciple of *Aspect-Oriented Programming* (AOP).

There are currently two primary ways of doing AOP: clear-box methods (like AspectJ [3] and Subject-Oriented Programming [4]) that require the source code of the to-be-AOP'ed primary system, mix aspects code into original base code, and output new code; and black-box methods (like Composition Filters [1] and OIF [7]) that wrap the to-be-AOP'ed components (functions, objects) in aspect wrappers. A more detailed discussion of the dimensions of aspect-oriented systems can be found in [8].

The great virtue of separation of concerns is that one can get people of differing expertise to work on different aspects of a system (who can even be not only physically but also temporally distant) and still meld their work together into a working system. Better AOP systems minimize the degree to which the programmers (particularly the programmers of the primary functionality) have to change their behavior to realize the benefits of AOP. It's a really nice bumper sticker to be able to say, "Just program like you always do, and we'll be able to add the aspects later. (And change your mind downstream about your policies, and we'll painlessly transform your code for that, too.)

4. Object Infrastructure Framework

At NASA Ames, we are working on a black-box Aspect-Oriented Programming system, the Object Infrastructure Framework (OIF). OIF realizes the following key ideas:

- **Intercepting communications.** OIF intercepts and manipulates communications among functional components, invoking appropriate "services" on these calls. Semantically, this is equivalent to wrapping or filtering [1] on both the client and server side of a distributed system. The next five points can be understood as describing the architecture of a flexible wrapping system.

- **Discrete injectors.** Our communication interceptors are first class objects: discrete components that have (object) identity and are invoked in a specific sequence. We call them *injectors*. In a distributed system, an ility may require injecting behavior on both the client and the server. (Figure 1 illustrates injectors on communication paths between components.) Injectors are uniform so we can build utilities to manipulate them.

- **Injection by object/method.** Each instance and each method on that object can have a distinct sequence of injectors.

- **Dynamic injection.** The injectors on an object/method are maintained dynamically and can, with the appropriate privileges, be added and removed. Examples of the application of dynamic configuration include placing debugging and monitoring probes on running applications and creating software that detects its own obsolescence and updates itself.

- **Annotations**. Injectors can communicate among themselves by adding annotations to the underlying requests of the procedure call mechanism.

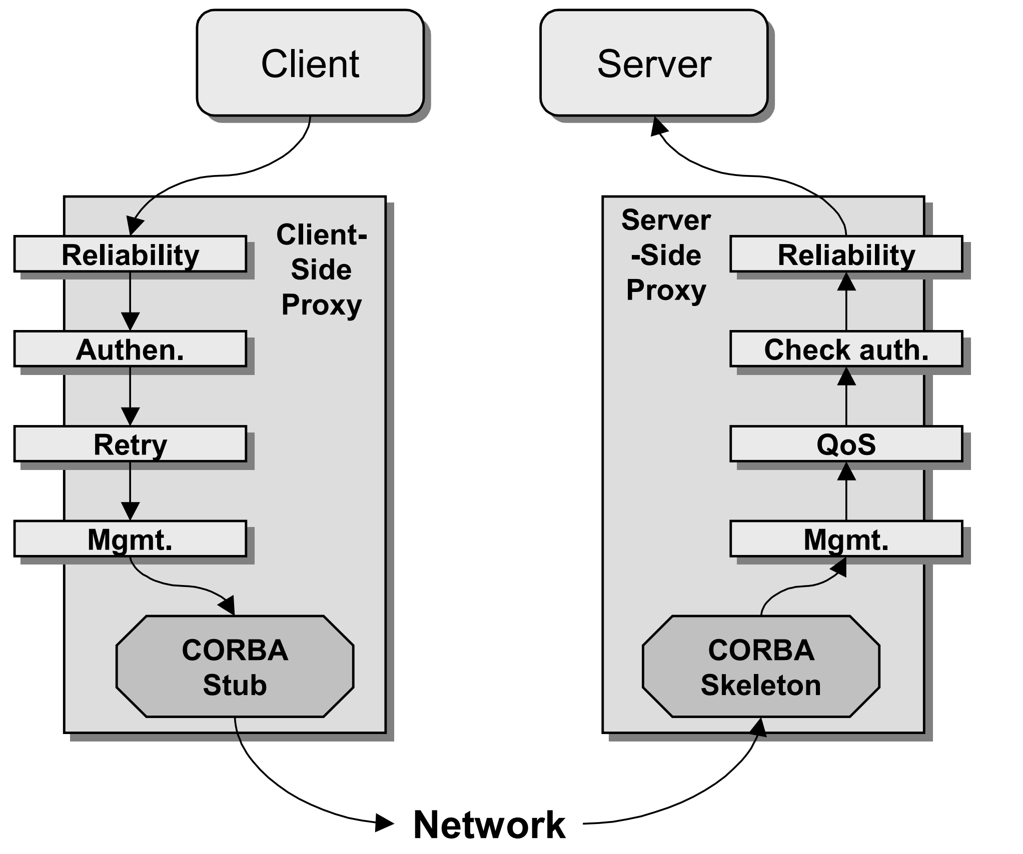


Figure 1 – Injectors on the Communication Path Between Components

- **Thread contexts.** Our goal is to keep the injection mechanism invisible to the functional components (or at least to those functional components that want to remain ignorant of it.) To allow clients and servers to communicate with the injector mechanism, the system maintains a "thread context" of annotations for threads, and copies between this context and the annotation context of requests. Thread contexts and annotations together provide the data space for communication between the application and injectors and among injectors. (Injectors generated by the same factory or industrial complex can also share a data space defined by their factory structure.)

- **High-level specification compiler.** To bridge the conceptual distance between abstract ilities and discrete sequences of injectors, we created a compiler from high-level specification of desired properties and ways to achieve these properties to default injector initializations.

OIF was developed to simplify distributed computing. We developed our prototype system for CORBA [10] components written in Java. CORBA is a distributed object technology (DOT) that presents remote objects as *proxy* objects in the local computing environment. Client-side proxies are responsible for encoding and forwarding a local call to the remote service; server-side proxies for decoding and returning the result. CORBA requires the description of an object's interface in its inter-lingual Interface Description Language (IDL). The CORBA IDL compiler then "compiles" that IDL into the code for the proxies in the desired execution language or languages. We implemented OIF by creating an alternative IDL compiler whose proxies both included calls to the proxy-specific sequence of injectors and maintained the request object / annotation / thread-context relationships.

A premise of the OIF work was that components are black boxes whose internal structure cannot be examined or manipulated. This contrasts with clear-box (source-language – level) approaches to AOP. Clear-box systems express aspects as code fragments to be woven together into a working program. This makes OIF injectors more likely to be reusable (as they are based on well-defined interfaces) but less powerful than clear-box mechanisms, which can (theoretically) be constructed as to perform arbitrary lower-level transformations on programs. In an environment like ISE, where the source for many applications is not available, black-box mechanisms are the clear choice.

Examples of injectors we have developed or are developing include are listed in Table 1. Several of these injectors are discussed in greater detail in [9].

5. Applying AOP to ISE

In this section, I present the outline of an Intelligent Synthesis Architecture for ISE. Critical to the ISE problem is incorporating and integrating legacy analysis tools. Such tools often exist in their own particular environments, take their own idiosyncratic inputs, and do who knows what else beyond their main functionality. The central themes of this architecture are (1) Legacy systems must be wrapped to fit into the ISA. The simpler we can make the manual part of this wrapping, the better. (2) The ISA serves as a repository of synthesis artifacts (including information about both the synthesis artifacts themselves and the process that synthesized the artifacts). (3) The ISA is an enabler of distributed

Table 1. Injectors

Ility	Injector	Action
Security	Authentication	Determines the identity of a user.
	Access control	Decides if a user has the privileges for a specific operation.
	Encryption	Encodes messages between correspondents.
	Intrusion detection	Recognizes attacks on the system.
Reliability	Replication	Replicates a database.
	Error retry	Catches network timeouts and repeats call.
	Rebind	Notices broken connections and opens connections to alternative servers.
	Voting	Transmits the same request to multiple servers (in sequence or parallel) combining the results by temporal or majority criteria.
	Transactions	Coordinates the behavior of multiple servers to all commit or fail together. Requires additional interface on application objects.
Quality of service	Queue-manager	Provides priority-based service.
	Side-door	Provides socket-based communication transparently to application.
	Futures	Provides futures transparently to the application.
	Caching	Caches results of invariant services.
Manageability	Logging	Reports dynamically on system behavior.
	Accounting	Reports to accounting system on incurred costs.
	Status	Accrues status information and reports when requested.
	Configuration management	Dynamically test for incompatible versions and automatically updates software.

computing and (4) the ISA provides a collection of synthesis and analysis services. The key ideas of this proposed ISA are

- **Reifying users, tools, and design artifacts.** Reifying is the idea of treating abstractions as things to which properties can be ascribed. Design artifacts are the inputs and outputs of tools. We call such things *entities*.

- **Recording *annotations* about entities.** For example, file XYZ is the output of running tool ABC on inputs G and H. This was done on May 15th by STU. It has the following permissions, precision, version, and so forth. We cannot anticipate all the annotations that users might want, so the set of annotations must be dynamic. We may also profitably apply the knowledge representation mechanisms of expert system building tools (e.g., default values, inheritance, access-oriented programming, and sentences in formal logics).

- **Invoking tools remotely and in scripts.** Since we know about the available tools and the annotations they impose on their outputs, and since tools are expressed as distributed objects, we can run tools (remotely) and can script together collections of tools.

- **Employing generic systems services within scripts and applications.** Tool scripts and application programs can take advantage of a number of system-provided services. The repository can be seen as be one such service.

I argue for a four layer architecture for the ISA (Figure 2). The lowest layer, the transport layer, enables secure remote invocation. Above it, a repository provides a database of synthesis artifacts and their attributes. The Common Services layer provides services on which every user of ISA can rely, and the Extended services layer uses the mechanisms of the repository and transport layer to provide optional additional facilities.

The obvious (from our point of view) choice for the transport layer is CORBA extended with OIF. By wrapping existing tools in CORBA wrappers, tools can be accessed by a distribution-transparent, software-bus mechanism. CORBA is the most mature of such architectures. In contrast with DCOM (Distributed Component Object Model), it runs on a large variety of operating systems and with programs compiled in many languages. In contrast with Java RMI (Remote Method Invocation), it is particularly tuned to legacy applications. And in contrast to the next generation of HTTP (HyperText Transfer Protocol), well, for one thing, it's already here.

OIF injectors on CORBA-wrapped tools and services can be used to

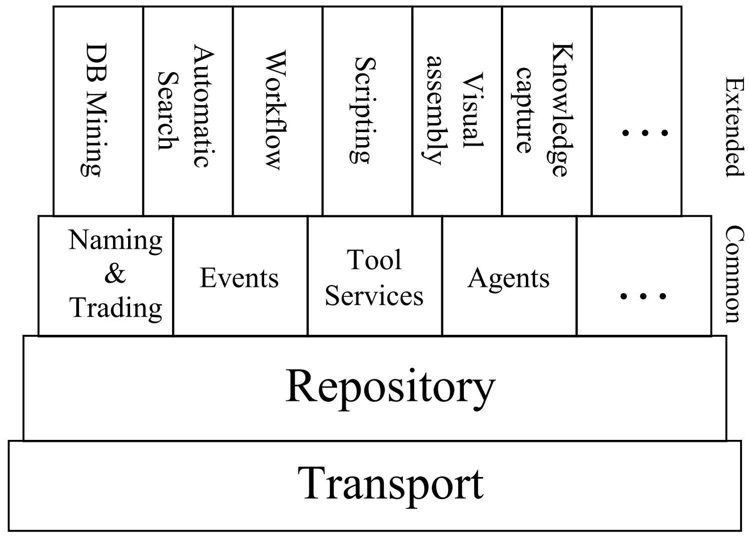


Figure 2. Intelligent Synthesis Architecture

- Maintain the annotations of artifacts created by running tools. For example, injectors can note the owner of a process and include in the repository the information about that owner, or store pointers to the inputs of a tool on the annotations of its outputs.

- Enforce complex, not-yet-anticipated access control rules on data, particularly as contractors form federations to deal with design subproblems.

- Implement automatic data set transformations to translate between representations.

- Supply alternative providers of the same service.

- Report on the status of jobs to distributed managers and debuggers.

- Support "session" environments reflecting user privileges downstream and carrying the user environment.

- Enable "long-lived" transactions needed by the design process.

- Obtain and assure the appropriate versions of datasets.

- Provide software redundancy and mobility, enabling moving computations and data for increased efficiency.

Each of these concepts can be realized by inserting the appropriate injectors on some or all of the methods of a tool or service. It is also the case that for each of these, we don't really know at this point what exactly we want done—what we mean by versions, access control restrictions, system management, transactions, and so forth. However, this is a strength of the injector approach, in that we can experiment and resolve these issues by the results of these experiments, rather than demanding omniscience at design time.

The Common Services layer would support services such as name serving, matchmaking, trading, events, system management, agent infrastructure, and tool wrapping. The Extended Services layer would support services such as repository mining, automatic design search, workflow automation (notifications and actions on events), visual assembly, and knowledge capture. I mention these more for completeness than for their relevance to AOP, though some might encode well as injectors.

Figure 3 lists the Java code for a simple *tracing* injector. This injector prints out the function being called and its arguments on entry and the result value on exit. It works on both clients and servers. The following points about this injector refer to the comment note numbers in the code

[1] Creating an injector involves creating two classes, the injector itself and a factory for creating instances of that injector. In using the system, the programmer refers to the injector factory; the run-time system calls on that factory to create injector instances when building CORBA proxies. Here we adopt the idiom of making the injector itself be an inner class of the factory.

[2] All injector factories have a single method, `createInjector`, which, given a proxy object, the name of the method on which the injector is to be applied, and a name-value list of properties, returns the injector object to be put on that proxy/method. Different patterns of injector sharing (for example, singletons, where all proxies share the same injector object; per method, where all the proxies of a given method share the same injector; and per instance, where each proxy gets its own injector) can be achieved through different implementations of the injector factory. The injector factory also serves as a locus for sharing information among injectors, such as a cache of network service quality data.

[3] In our example, the factory simply creates a new `Tracer` for each proxy/method. (This is simple but wasteful, as the `Tracer` has no object state. A more efficient implementation would be a singleton.)

[4] The injector itself (the class `Tracer`) has two methods, a constructor and `exec`.

[5] The constructor is called on creating the proxy (and given the proxy, the method being applied and a property list.)

[6] Method `exec` is called operationally, whenever a request passes through the injector. It is given the request object and can throw an `InjectorException`.

```
package tracing;
import oif.framework.*;

public class TracingInjectorFactory implements InjectorFactoryType{      /* [1] */

    public InjectorType createInjector(ProxyType proxy,                    /* [2] */
                                       String methodName,
                                       OIFProperties props) {
        return new Tracer(proxy, methodName, props);                       /* [3] */
    }

    class Tracer extends InjectorBase{

        public Tracer (ProxyType proxy,                                    /* [4] */
                       String methodName,
                       OIFProperties props) {
            super(proxy, methodName, props);                               /* [5] */
        }

        public void exec(OIFRequestType req) throws InjectorException{  /* [6] */
            System.out.println(req.getProxy( ).getClass( ).getName( ) + /* [7] */
                               " Op: " + req.getOperation( ) + "\n" );
            req.listArgs();
            doNext(req);                                                   /* [8] */
            System.out.println( req.getProxy( ).getClass( ).getName( ) +
                               " Op: " + req.getOperation( ) + "\n" );
            System.out.println("Result: " + req.getResult( ));
        }
    }
}
```

Figure 3: The Tracing Injector

[7] The request object stores several things, such as the underlying proxy object (`req.getProxy()`), the operation (method name) being called (`req.getOperation()`), the operation arguments (which are printed by `req.listArgs()`), and the return value (`req.getResult( )`). The injector has read/write access to all of these.

[8] Injectors implement a continuation structure: Each injector is responsible for invoking the remaining injectors by calling `doNext()` on the request object. This organization enables architectures where the rest of the computation can be invoked in several different places in the code (which might be the case if we wanted to do different things on the client and the server), repeatedly invoked (as is done in a retrying error recovery) or not invoked at (as would be done by a cache injector on finding the requested value already in its cache.)

OIF maintains a Proxy Initialization Table (PIT) that maps, for each proxy class, each method in that class, and the client/server distinction, a sequence of injector factories. When a proxy is created, those factories are invoked to create a (correspondingly ordered) sequence of injectors for each method on the proxy. (Operations exist to dynamically modify both the PIT and the injectors on a proxy. The system developer can choose whether to expose those operations, and, if exposed, the security mechanisms to use to protect them.)

The system developer can specify the (initial) configuration of injectors on proxy methods through the specification language Pragma. Figure 4 shows the Pragma specification for a simple two-injector system. We note the following points about that specification:

[1] Pragma statements are collected into “policies.” Though its not illustrated in the example, policies can include other policies.

[2] Import statements become import statements in the Java output. Like Java, import statements are used to abbreviate complex names.

[3] The user declares abstract “ilities” (concerns). This program has two concerns, `Context` and `QualityOfService`. `Context` implements the notion of copying the application thread context through the request annotations and on to the service thread context. `QualityOfService` speaks to providing better service to more

```
policy bank is                                        /* [1] */

    import Bank;                                      /* [2] */
    import QManager;
    import ContextPkg;

    ility Context,                                    /* [3] */
          QualityOfService;

    var priority : int = {1};                         /* [4] */

    define copyContext for Context as                 /* [5] */
        client ContextInjectorFactory do first,
        server ContextInjectorFactory do last;

    define queueing for QualityOfService as           /* [6] */
        server QManager.QueueManagerInjectorFactory;

    for Context do copyContext;                       /* [7] */

    for QualityOfService                              /* [8] */
        on open
        in AccountManager
      do queueing;

    for QualityOfService                              /* [9] */
        on balance
        in Account
      do queueing;

end;
```

Figure 4: Pragma for quality-of-service queues

important requests. There will be at most one way of achieving each ility on any proxy/method.

[4] Here we declare the annotation `priority`, and specify that it is an `int` with a default value of 1. We will use the priority of a request to determine its service level.

[5] One way of achieving the `Context` ility is through `copyContext`. `copyContext` works by having the `ContextInjector` as the first injector on the client side, and the last injector on the server side.

[6] One way of achieving `QualityOfService` is with `queuing`. `queuing` works by having the `QueueManagerInjector` on the server side.

[7] We achieve the `Context` ility by using `copyContext`. Since this statement is not modified by particular interfaces or methods, its done on all interfaces and methods.

[8] We achieve `QualityOfService` on the `open` method of proxies supporting the `AccountManager` interface by doing `queuing`.

[9] Similarly, we achieve `QualityOfService` on the `balance` method of `Account` objects through `queuing`.

Executing the Pragma compiler on this specification and the application IDL produces a Java program to initialize the PIT.

6. CONCLUDING REMARKS

The Object Infrastructure Framework provides a mechanism for performing Aspect-Oriented Programming at the component level. We have discussed how the OIF mechanism could be applied to the development of an architecture for an Intelligent Synthesis Environment .

ACKNOWLEDGMENTS

My thanks to Diana Lee and Tarang Patel for comments on the drafts of this paper.

REFERENCES

[1] Daniel S. Goldin, Samuel L. Venneri, and Ahmed K. Noor, "Beyond Incremental Change," *IEEE Computer* ***31***, 31–39, October 1998.

[2] Padmanabh Dabke, "Enterprise Integration via CORBA-Based Information Agents," *IEEE Internet Computing* ***3***, 49–57, September 1999.

[3] Martin Hardwick, David L. Spooner, Tom Rando, and K. C. Morris, "Data Protocols for the Industrial Virtual Enterprise," *IEEE Internet Computing* ***1,*** 20–29, January 1997.

[4] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-Oriented Programming," In *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP), Finland. Berlin: Springer-Verlag LNCS 1241. June 1997.

[5] William Harrison, and Harold Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)," Proc. OOPSLA '93. *ACM SIGPLAN Notices* ***28***, 411–428, October 1993.

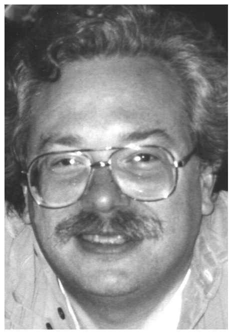
[6] Mehmet Aksit, and Bedir Tekinerdogan, "Solving the Modeling Problems of Object-Oriented Languages by Composing Multiple Aspects Using Composition Filters," AOP '98 workshop position paper, 1998. http://wwwtrese.cs.utwente.nl/Docs/Tresepapers/FilterAspects.html

[7] Robert E. Filman, Stuart Barrett,. Diana D. Lee., and Ted Linden, "Inserting Ilities by Controlling Communications," *Comm. ACM,* in press. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/oif-cacm-final.pdf

[8] Robert E. Filman, and Daniel P. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness," Workshop on Advanced Separation of Concerns, OOPSLA 2000, Oct. 2000, Minneapolis. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/aop-is.pdf

[9] Robert E. Filman, David J. Korsmeyer, and Diana D. Lee, "A CORBA Extension for Intelligent Software Environments," *Advances in Engineering Software* ***3,*** 727–732, 2000. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/williamsburg-print-final.pdf

[10] Jon Siegel, *CORBA: Fundamentals and Programming*. New York: Wiley, 1996.



Robert E. Filman *is a Senior Scientist at the Research Institute for Advanced Computer Science at NASA Ames Research Center, working on creating frameworks for developing distributed applications. Prior to coming to NASA, Dr. Filman worked in the research groups of Lockheed Martin Missiles and Space, Intellicorp and Hewlett-Packard Laboratories, and on the faculty of the Computer Science Department at Indiana University, Bloomington. He is Associate Editor-in-Chief of IEEE Internet Computing and is on the editorial board of the International Journal of Artificial Intelligence Tools. He is the author (with Daniel P. Friedman) of Coordinated Computing: Tools and Techniques for Distributed Software (McGraw-Hill). Dr. Filman received his B. S. (Mathematics), and M.S. and Ph. D. (Computer Science) from Stanford University.*